Clase 4 — 11.11.25

#IntelliJIDEA #JAVA

👨‍🏫 **Profesor:** Álvaro García Gutiérrez
📘 **Acceso a Datos**
🗓️ **Clase 4 — 11/11/2025**
🎯 **Tema: Ficheros Binarios en Java | writeUTF / registros fijos / acceso directo | Cuestionario**

1️⃣ Introducción

En esta clase seguimos profundizando en **RandomAccessFile**, una herramienta muy potente que permite:

- Leer y escribir datos binarios.
- Controlar la posición exacta del puntero con `seek()`.
- Crear **registros de tamaño variable** (UTF) o **registros de tamaño fijo** (id + nombre, etc.).

El objetivo es entender:

- Cuándo se puede usar **acceso aleatorio real**.
- Por qué `writeUTF/readUTF` **no sirven para acceso directo**, solo secuencial.
- Cómo construir nuestros propios **registros de tamaño fijo** para simular sistemas más cercanos a bases de datos.

La clase se divide en varios bloques:

1. **Ejercicio 3** → Cadenas con `writeUTF` y lectura secuencial.
2. **Ejercicio 4** → Registros fijos (id + nombre fijo) y acceso directo con `seek()`.
3. **Ejercicio 5** →

2️⃣ Ejercicio 3 — Cadenas cortas con writeUTF/readUTF

¿Qué hace writeUTF realmente?

`writeUTF(texto)` NO escribe solo caracteres. Escribe:

1. **2 bytes** → longitud (en bytes) del texto en Modified UTF-8.
2. **N bytes** → contenido codificado en UTF-8.

Esto significa:

- Cada cadena ocupa **tamaño distinto** en el fichero.
- NO existe una fórmula para saltar al “registro 2” directamente.
- La única forma fiable de localizar el siguiente registro es **leer el anterior**.

Por eso el profesor enfatiza:

> “Escribir y leer secuencialmente está bien, pero no se puede hacer acceso aleatorio con UTF.”

2.1 Código completo explicado

```java
package tarea.obligatoria.pkg3.random.acces.file;

import java.io.IOException;
import java.io.RandomAccessFile;

/**
 * Ejercicio 3 — Cadenas cortas con writeUTF/readUTF.
 *
 * writeUTF guarda la longitud de la cadena + sus bytes en formato UTF.
 * readUTF lee el mismo formato y recupera el String tal y como se escribió.
 */
public class Ejercicio3 {

    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("nombres.bin", "rw")) {

            // Al comenzar, dejamos el archivo vacío para evitar errores de restos
anteriores.
            raf.setLength(0);

            // ---- 1) Escritura secuencial de tres nombres ----
            // Aquí podríamos pedirlo por pantalla, pero el profesor los escribe
directamente.
            raf.writeUTF("Ana");
            raf.writeUTF("Luis");
            raf.writeUTF("Marta");

            // El puntero queda al final tras las escrituras.
            // Para leer debemos volver explícitamente al inicio.
            raf.seek(0);

            // ---- 2) Lectura secuencial ----
            String n1 = raf.readUTF();   // Lee longitud + bytes → reconstruye String
            System.out.println(n1);

            String n2 = raf.readUTF();
            System.out.println(n2);

            String n3 = raf.readUTF();
            System.out.println(n3);

            // Mostrar dónde quedó el puntero
            long finalPos = raf.getFilePointer();
            System.out.println("getFilePointer() al final = " + finalPos);

            // ---- Explicación crítica ----
            // NO podemos saltar directamente al segundo nombre con seek(n).
            // writeUTF escribe longitud variable → no existe un tamaño fijo para
calcular posiciones.
        }
    }
}
```

2.2 Puntos críticos del Ejercicio 3

✔ `setLength(0)`

Vacía el fichero para empezar limpio.

✔ `seek(0)`

Obligatorio para volver a leer desde el principio.

✔ writeUTF = longitud + bytes (tamaño variable)

Cada nombre ocupa espacio distinto ⇒ NO se puede usar acceso aleatorio.

✔ Los datos se leen EXACTAMENTE en el mismo orden en que se escribieron

Si no se mantiene el orden, el puntero queda desincronizado.

3️⃣ Ejercicio 4 — Registros fijos (id + nombre fijo)

Este ejercicio introduce un concepto fundamental:

> Para poder usar acceso aleatorio real con RandomAccessFile, los registros deben tener **tamaño fijo**.

Diseñamos un registro con:

Campo	Tipo	Tamaño
id	int	4 bytes
nombre	10 chars	20 bytes
Total	—	**24 bytes**

3.1 Código completo explicado

```
package tarea.obligatoria.pkg3.random.acces.file;

import java.io.IOException;
import java.io.RandomAccessFile;

public class Ejercicio4 {

    // Nombre fijo de 10 caracteres → 20 bytes
    static final int NCHARS = 10;

    // Registro fijo: id (4 bytes) + nombre (20 bytes) = 24 bytes
    static final int BYTES_REG = 4 + 2 * NCHARS;

    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("personas.dat", "rw")) {

            raf.setLength(0); // Muy importante: limpiamos el archivo
```

```
            // ---- 1) Escribir 3 registros de ejemplo ----

            // Registro 0 (offset 0)
            raf.seek(0 * BYTES_REG);
            escribirRegistro(raf, 1, "Ana");

            // Registro 1 (offset 24)
            raf.seek(1 * BYTES_REG);
            escribirRegistro(raf, 2, "Bernardo");

            // Registro 2 (offset 48)
            raf.seek(2 * BYTES_REG);
            escribirRegistro(raf, 3, "Clara");

            // ---- 2) Leer directamente el tercer registro ----
            int indice = 2;
            long offset = indice * BYTES_REG;

            raf.seek(offset);               // Saltamos directo al registro 2
            int id = raf.readInt();         // id = 3
            String nombre = leerNombreFijo(raf, NCHARS);

            System.out.println("Tercer registro -> id=" + id + ", nombre=" + nombre);
        }
    }

    // Escribir registro en la posición actual del puntero
    static void escribirRegistro(RandomAccessFile raf, int id, String nombre) throws
IOException {
        raf.writeInt(id); // 4 bytes
        escribirNombreFijo(raf, nombre, NCHARS);
    }

    // Escribir nombre de longitud fija (rellenando con espacios si hace falta)
    static void escribirNombreFijo(RandomAccessFile raf, String s, int len) throws
IOException {
        if (s == null) s = "";
        if (s.length() > len) {
            s = s.substring(0, len); // recortar si es más largo
        }
        String ajustado = String.format("%-" + len + "s", s); // completar con
espacios
        for (int i = 0; i < len; i++) {
            raf.writeChar(ajustado.charAt(i)); // cada char = 2 bytes
        }
    }

    // Leer un nombre de longitud fija usando un buffer
    static String leerNombreFijo(RandomAccessFile raf, int len) throws IOException {
        StringBuilder sb = new StringBuilder(len); // más eficiente que concatenar
Strings
        for (int i = 0; i < len; i++) {
            sb.append(raf.readChar()); // readChar lee 2 bytes
        }
```

```
            return sb.toString().trim();
        }
    }
```

3.2 Puntos críticos del Ejercicio 4

✔ Diseño del registro fijo

- El tamaño de cada registro es **conocido y constante**:
  `BYTES_REG = 24 bytes`.

Esto permite calcular el offset:

```
offset = índice * 24
```

✔ writeChar / readChar → 2 bytes exactos por carácter

Por eso los nombres fijos ocupan:

```
10 chars × 2 bytes = 20 bytes
```

✔ String.format() para rellenar con espacios

`String.format("%-10s", "Ana")` → `"Ana "`.

✔ Saltar directamente a un registro funciona porque:

- Cada registro ocupa 24 bytes.
- El offset siempre cae en una posición válida.

Ejemplo:

> Registro 2 → 2 × 24 = byte 48

✔ Buffer de lectura (`StringBuilder`)

- Leer todos los chars a un buffer es más eficiente.
- Después se hace `trim()` para quitar los espacios de relleno.

4️⃣ Diferencia conceptual entre Ejercicio 3 y 4

Aspecto	Ejercicio 3	Ejercicio 4
Forma de escribir texto	writeUTF	writeChar (fijo)
Tamaño del registro	Variable	**Fijo**
Se puede calcular el offset	❌ No	✔ Sí
Se puede acceder directamente al registro N	❌ No	✔ Sí
Necesita leer secuencialmente	✔ Sí	❌ No
Uso típico	cadenas simples	estructuras tipo BD

5️⃣ Conclusión de los ejercicios anteriores

- `writeUTF` es potente para almacenar texto, pero NO sirve para acceso aleatorio porque los registros son variables.
- Para acceso directo necesitamos **registros fijos** → misma estructura y mismo tamaño siempre.
- `seek()` permite reposicionar el puntero con precisión, pero solo es útil si sabemos exactamente dónde empieza cada registro.
- `setLength(0)` debe usarse siempre que queramos "resetear" un fichero binario antes de reescribirlo.
- RandomAccessFile es una herramienta básica para modelar estructuras internas de bases de datos simples o ficheros indexados.

📝 Tarea Obligatoria 4 (Ejercicio 5)

Actualización de un campo dentro de un registro fijo en RandomAccessFile

Este ejercicio profundiza en el uso de **RandomAccessFile** para acceder y modificar **solo un campo** dentro de un registro estructurado.

Trabajamos sobre un archivo binario `personas.dat`, donde cada registro tiene un formato **fijo**.

1️⃣ Estructura del registro (muy importante)

Cada persona ocupa exactamente **48 bytes** en el fichero:

Campo	Tipo	Tamaño
id	int	4 bytes
nombre	20 chars	40 bytes (cada char = 2 bytes con writeChar)
edad	int	4 bytes

TOTAL: 48 bytes por registro

Esto nos permite posicionarnos exactamente en cualquier registro usando:

```
offset = índiceRegistro * 48
```

2️⃣ Código completo comentado por bloques

```
package tarea.obligatoria.pkg4.actualizar.campo.en.random.access.file;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

/**
 * TAREA OBLIGATORIA 4 – Actualizar un campo dentro de un registro fijo
 *
 * Formato del registro (48 bytes):
 *   - id     : int (4 bytes)
 *   - nombre : 20 chars → 20 * 2 bytes = 40 bytes
 *   - edad   : int (4 bytes)
 *
```

```java
 * TOTAL: 4 + 40 + 4 = 48 bytes por registro
 *
 * El objetivo es modificar SOLO la edad de un registro, sin tocar id ni nombre.
 */

public class TAREAOBLIGATORIA4ActualizarCampoEnRandomAccessFile {

    // Tamaño fijo del nombre en caracteres (se convertirá en 40 bytes)
    private static final int NCHARS = 20;

    // Tamaño total del registro en bytes
    private static final int BYTES_REG = 4 + (2 * NCHARS) + 4;

    public static void main(String[] args) throws IOException {

        try (Scanner sc = new Scanner(System.in);
             // Abrimos o creamos personas.dat en lectura/escritura
             RandomAccessFile raf = new RandomAccessFile("personas.dat", "rw")) {

            // ========================================================================
            // 1) Si el fichero está vacío, escribimos 3 registros por defecto
            // ========================================================================
            if (raf.length() == 0) {
                // Cada vez que llamamos a escribirRegistro(), el puntero avanza 48
bytes
                escribirRegistro(raf, 1, "Ana",   30);  // Registro 0 → offset 0
                escribirRegistro(raf, 2, "Luis",  25);  // Registro 1 → offset 48
                escribirRegistro(raf, 3, "Maria", 40);  // Registro 2 → offset 96
            }

            // ========================================================================
            // 2) Pedimos al usuario el id a actualizar y la nueva edad
            // ========================================================================
            System.out.print("Introduce ID a actualizar: ");
            int idBuscado = sc.nextInt();

            System.out.print("Nueva edad: ");
            int nuevaEdad = sc.nextInt();

            // ========================================================================
            // 3) Buscar secuencialmente el registro que tenga ese id
            // ========================================================================

            long numRegs = raf.length() / BYTES_REG;  // Nº total de registros
            long regInicio = -1;                      // Offset del registro
encontrado

            // Recorremos todos los registros uno por uno
            for (int i = 0; i < numRegs; i++) {

                // Cálculo del offset del registro i
                long offset = (long) i * BYTES_REG;   // Ej: reg 2 → 2*48 = 96

                // Posicionamos el puntero al inicio del registro i
```

```
                raf.seek(offset);

                // Leemos el id del registro (4 bytes)
                int id = raf.readInt();

                if (id == idBuscado) {
                    // Coincidencia → guardamos el inicio del registro
                    regInicio = offset;
                    break;
                }

                // Si no coincide, NO hace falta saltar manualmente el registro.
                // En la siguiente iteración recalculamos offset y usamos
seek(offset).
            }

            // Si no hemos encontrado el registro, avisamos
            if (regInicio == -1) {
                System.out.println("ID no encontrado.");
                return;
            }

            // ================================================================
            // 4) Actualizar SOLO el campo "edad" dentro del registro encontrado
            // ================================================================

            // Dentro del registro: [id(4B)] [nombre(40B)] [edad(4B)]
            // → edad empieza exactamente en offset = inicio + 4 + 40 = inicio + 44
            long offsetEdad = regInicio + 4 + (2L * NCHARS); // 4 + 40 = 44

            // Movemos el puntero SOLO al campo edad
            raf.seek(offsetEdad);

            // Sobrescribimos los 4 bytes de edad sin tocar nada más
            raf.writeInt(nuevaEdad);

            // ================================================================
            // 5) Verificar lectura del registro actualizado
            // ================================================================

            // Volvemos al inicio del registro para leerlo entero
            raf.seek(regInicio);

            int id = raf.readInt();                     // Leer id
            String nombre = leerNombreFijo(raf, NCHARS); // Leer nombre (20 chars)
            int edad = raf.readInt();                   // Leer edad actualizada

            System.out.println(
                    "Actualizado: id=" + id +
                    ", nombre=" + nombre +
                    ", edad=" + edad
            );
        }
    }
```

```java
    // ========================================================================
    // MÉTODOS AUXILIARES
    // ========================================================================

    /**
     * Escribe un registro completo en la posición actual del puntero.
     */
    private static void escribirRegistro(RandomAccessFile raf, int id, String nombre,
int edad) throws IOException {

        raf.writeInt(id);                          // Escribe id (4 bytes)
        escribirNombreFijo(raf, nombre, NCHARS);  // Escribe nombre (20 chars → 40
bytes)
        raf.writeInt(edad);                        // Escribe edad (4 bytes)
    }

    /**
     * Escribe un nombre EXACTAMENTE de 'len' caracteres.
     * Si es corto → rellena con espacios.
     * Si es largo → recorta.
     */
    private static void escribirNombreFijo(RandomAccessFile raf, String s, int len)
throws IOException {

        if (s == null) s = "";
        if (s.length() > len) s = s.substring(0, len);

        // "%-20s" → rellena a la derecha con espacios hasta 20 chars
        String fijo = String.format("%-" + len + "s", s);

        // writeChar escribe CADA caracter en 2 bytes
        for (int i = 0; i < len; i++) {
            raf.writeChar(fijo.charAt(i));
        }
    }

    /**
     * Lee un nombre fijo de 'len' caracteres.
     * readChar lee 2 bytes → perfecto para este formato.
     */
    private static String leerNombreFijo(RandomAccessFile raf, int len) throws
IOException {

        StringBuilder sb = new StringBuilder(len);

        for (int i = 0; i < len; i++) {
            // Cada readChar consume 2 bytes del fichero
            sb.append(raf.readChar());
        }

        // Eliminamos los espacios usados como relleno
        return sb.toString().trim();
```

```
    }
}
```

Apertura del fichero y valores constantes

```
static final int NCHARS = 20;                // Tamaño fijo del nombre
static final int BYTES_REG = 4 + 2*NCHARS + 4;  // 48 bytes por registro
```

- `NCHARS = 20` → todos los nombres ocuparán 20 caracteres exactos.
- Como cada `char` ocupa **2 bytes**, el nombre ocupa 40 bytes en total.
- El registro completo: **4 (id) + 40 (nombre) + 4 (edad) = 48 bytes**.

3️⃣ Escritura inicial de registros (solo si el fichero está vacío)

```
if (raf.length() == 0) {
    escribirRegistro(raf, 1, "Ana",   30);
    escribirRegistro(raf, 2, "Luis",  25);
    escribirRegistro(raf, 3, "Maria", 40);
}
```

- Si `personas.dat` está vacío, creamos tres registros básicos.
- Muy útil para no tener que introducirlos manualmente cada vez.
- Cada llamada a `escribirRegistro()` deja el puntero al final del registro escrito.

4️⃣ Solicitud de datos al usuario

```
System.out.print("Introduce ID a actualizar: ");
int idBuscado = sc.nextInt();

System.out.print("Nueva edad: ");
int nuevaEdad = sc.nextInt();
```

Simple lectura del **id a buscar** y la **nueva edad**.

5️⃣ Búsqueda secuencial del registro por id

Aquí está la parte más crítica del ejercicio.

```
long numRegs = raf.length() / BYTES_REG;  // nº total de registros
long regInicio = -1;
```

- Dividimos el tamaño del fichero entre 48 bytes para saber cuántos registros hay.
- `regInicio` almacenará el **offset exacto** donde empieza el registro buscado.

🔍 Recorrido registro por registro

```
for (int i = 0; i < numRegs; i++) {
```

```
    long offset = (long) i * BYTES_REG;
    raf.seek(offset);

    int id = raf.readInt();               // Lee solo el id

    if (id == idBuscado) {
        regInicio = offset;
        break;
    }
}
```

¿Qué ocurre aquí?

- **offset** calcula la posición exacta del registro `i` → `i * 48`.
- `seek(offset)` mueve el puntero al **comienzo del registro**.
- `raf.readInt()` lee solo el id (4 bytes); el nombre y edad NO se leen todavía.
- Si coincide, guardamos la posición y salimos del bucle.

No hace falta saltar manualmente el registro, porque el siguiente `offset` en la siguiente iteración ya recoloca el puntero.

6️⃣ Comprobación de que el id existe

```
if (regInicio == -1) {
    System.out.println("ID no encontrado.");
    return;
}
```

Evita errores si el usuario introduce un id inexistente.

7️⃣ Cálculo del offset exacto del campo "edad"

Esta es la parte clave del ejercicio.

```
long offsetEdad = regInicio + 4 + (2L * NCHARS);
```

Interpretación:

- `regInicio` = inicio del registro
- `+ 4` → saltamos el campo id
- `+ 2 * NCHARS = 2*20 = 40` → saltamos los 20 chars del nombre

Total: regInicio + 44
→ Aquí empieza el campo `edad` (4 bytes).

8️⃣ Escritura de la nueva edad (sin tocar nada más)

```
raf.seek(offsetEdad);   // Posicionar el puntero justo en edad
raf.writeInt(nuevaEdad);
```

Esto sobrescribe únicamente el campo **edad**, respetando:

- id
- nombre

9️⃣ Verificación leyendo el registro completo

```
raf.seek(regInicio);

int id = raf.readInt();
String nombre = leerNombreFijo(raf, NCHARS);
int edad = raf.readInt();
```

- Volvemos al inicio del registro actualizado.
- Leemos los 48 bytes completos.
- Mostramos todo para confirmar el cambio.

🔧 Funciones auxiliares

escribirRegistro()

Escribe un registro completo en la posición actual del puntero:

```
raf.writeInt(id);
escribirNombreFijo(raf, nombre, NCHARS);
raf.writeInt(edad);
```

escribirNombreFijo()

Rellena o recorta a exactamente 20 chars:

- Si el nombre es corto → añade espacios.
- Si es largo → lo recorta.
- Escribe cada char con `writeChar()` (2 bytes).

leerNombreFijo()

Lee exactamente 20 `char` y hace `.trim()` al final.

✔ Conclusión del ejercicio

Este ejercicio demuestra cómo:

- Gestionar **registros de longitud fija** en binario.
- Calcular posiciones exactas dentro del archivo usando **offsets**.
- Actualizar **solo un campo** sin afectar al resto del registro.
- Usar correctamente `seek()`, `writeInt()`, `readInt()` y lectura de cadenas fijas con `writeChar/readChar`.

Es un patrón típico en **ficheros de acceso aleatorio**, muy usado históricamente en bases de datos primitivas y sistemas de almacenamiento estructurado.

🧩 Apuntes ampliados — Tarea Obligatoria 4 (Ejercicio 5)

Actualización de un campo dentro de un registro fijo en RandomAccessFile

---

1️⃣ Formato del registro (48 bytes)

Cada persona ocupa **exactamente 48 bytes**:

Campo	Tipo	Tamaño
id	int	4 bytes
nombre	20 chars	40 bytes (20 × 2 bytes)
edad	int	4 bytes

Importante:
Este tamaño **no puede cambiar jamás** o el acceso aleatorio dejará de funcionar correctamente.

2️⃣ Inicialización del fichero

Si el fichero está vacío (`raf.length() == 0`), creamos registros iniciales con:

- `escribirRegistro()` → escribe id + nombre fijo + edad
- Cada llamada avanza 48 bytes automáticamente (porque escribe 48 bytes)

3️⃣ Entrada del usuario (id + nueva edad)

Leemos dos valores:

- `idBuscado` → registro a localizar
- `nuevaEdad` → el valor que sustituirá al campo edad dentro del registro

---

4️⃣ Búsqueda secuencial del registro

La búsqueda NO se hace usando índices lógicos (“pos 1, pos 2…”).
Se hace directamente sobre el **fichero binario**:

```
long numRegs = raf.length() / BYTES_REG;
```

Esto responde a la primera pregunta clave:

✔ ¿Cómo sé cuántos registros hay?

Dividiendo el tamaño total de bytes entre el tamaño del registro:

```
numRegs = tamañoFichero / 48
```

Sin esta división no sabrías cuántas veces iterar para recorrer el fichero.

✔ ¿Por qué uso `long regInicio = -1` ?

Porque:

- Necesitamos almacenar dónde empieza el registro encontrado.
- Si NO se encuentra, necesitaremos detectarlo de forma clara.

Los offsets válidos son `0, 48, 96, 144…` : **nunca negativos**.

Por eso `-1` funciona como “marcador de NO encontrado”.

5️⃣ Cálculo de offsets dentro del registro

Cuando encontramos el registro:

```
regInicio = offsetBuscado
```

Ahora necesitamos localizar el campo concreto **edad**, que está al final del registro:

```
[id: 4 bytes][nombre: 40 bytes][edad: 4 bytes]
```

Por tanto:

```
long offsetEdad = regInicio + 4 + (2L * NCHARS);
```

✔ ¿Por qué esta fórmula?

- `4` → saltamos el id
- `2 * NCHARS` → saltamos el nombre (20 chars × 2 bytes)
- Ahora el puntero está EXACTAMENTE sobre el campo edad

Desde ahí, escribir:

```
raf.writeInt(nuevaEdad);
```

sobrescribe SOLO los 4 bytes de ese campo.

Nada más. Nada menos.

6️⃣ Verificación final del registro

Para asegurarnos de que la modificación es correcta:

1. Volvemos al inicio del registro:
   ```
   raf.seek(regInicio);
   ```
2. Leemos los 48 bytes completos en orden:
   - id → `readInt()`
   - nombre → `leerNombreFijo()`
   - edad → `readInt()`

Esto garantiza que el registro ha sido correctamente modificado y está bien formado.

7️⃣ Puntos críticos del ejercicio

✔ Mantener un tamaño fijo de registro

Si los registros no ocuparan siempre 48 bytes:

- no podrías calcular offsets,
- no podrías saltar entre registros,
- se descuadraría toda la estructura.

✔ Siempre leer en el mismo orden que se escribe

RandomAccessFile **no almacena formato**, solo bytes.
Si escribes:

```
int → nombre fijo → edad
```

debes leer:

```
int → nombre fijo → edad
```

Modificar el orden rompe completamente el fichero.

✔ No usar writeUTF/readUTF en registros fijos

Porque writeUTF escribe:

- longitud (2 bytes)
- bytes UTF (tamaño variable)

Y eso hace que los campos no tengan tamaño constante.

✔ No hace falta saltar manualmente el nombre y edad durante la búsqueda

¿Por qué?

Porque el for hace:

```
offset = i * 48
seek(offset)
readInt()  // solo leemos el id
```

y en la siguiente iteración:

```
offset = (i+1) * 48
seek(offset)
```

El puntero SIEMPRE vuelve al inicio del siguiente registro gracias a `seek()`.

8️⃣ Preguntas importantes que debemos hacernos

❓ ¿Cuántos registros hay en el fichero?

Porque `numRegs = length / BYTES_REG`.

❓ ¿Por qué necesito `numRegs`?

Para iterar exactamente sobre todos los registros del fichero.

❓ ¿Por qué `regInicio = -1`?

Para detectar si NO se ha encontrado el registro.

❓ ¿Dónde empieza el campo edad dentro de un registro?

En el byte:

```
regInicio + 4 + 40 = regInicio + 44
```

❓ ¿Qué pasa si escribo más caracteres que NCHARS?

Se recortan.
Si escribo menos, se rellenan con espacios.

❓ ¿Qué ocurre si leo un campo en orden incorrecto?

El puntero queda desalineado → leerás basura binaria.

❓ ¿Por qué usar StringBuilder en leerNombreFijo?

Porque es muchísimo más eficiente que concatenar Strings dentro del bucle.

Tarea Complementaria 3

Apertura: martes, 11 de noviembre de 2025, 17:00

Cierre: domingo, 30 de noviembre de 2025, 23:00

Índice por id en archivo aparte (acceso directo por clave)

Diseñar un índice auxiliar en un archivo separado para ir directo al registro por su id, evitando recorrer todo el fichero de datos.

Para ello:

• Recorre personas.dat (Tarea obligatoria 4) y construye personas.idx: por cada registro en posición i, escribe (id, i*48L) en el índice.

• Implementa una búsqueda por id:

➢ Abre personas.idx y recórrelo buscando el id (lineal para empezar).

➢ Cuando lo encuentres, salta a offset en personas.dat y lee solo ese registro.

Personas.dat (datos): mismo registro de 48B de la Tarea obligatoria 4.

Personas.idx (índice): pares (id:int, offset:long) → 12 bytes por entrada

(4B + 8B). El offset es el byte dentro de personas.dat donde empieza el registro (i* 48).

```
package tarea.complementaria.pkg3.random.access.file.con.indice;

import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

/**
 * Tarea complementaria 3:
```

```java
 * Uso de RandomAccessFile con un fichero de índice auxiliar.
 *
 * personas.dat  -> fichero de DATOS con registros FJOS de 48 bytes:
 *                  id (int, 4 B) + nombre (20 chars, 40 B) + edad (int, 4 B)
 *
 * personas.idx  -> fichero de ÍNDICE con entradas de 12 bytes:
 *                  id (int, 4 B) + offset (long, 8 B)
 *
 * El flujo del programa es:
 *   1) Nos aseguramos de que personas.dat tenga algunos registros de ejemplo.
 *   2) Recorremos personas.dat y reconstruimos personas.idx desde cero.
 *   3) Pedimos un id por teclado y lo buscamos en personas.idx.
 *   4) Si lo encontramos, usamos el offset para saltar DIRECTAMENTE
 *      al registro correspondiente en personas.dat y lo mostramos.
 */
public class TareaComplementaria3RandomAccessFileConIndice {

    // --------- Formato del registro en personas.dat ---------

    // Número de caracteres fijos para el campo nombre (20 chars -> 40 bytes)
    static final int NCHARS = 20;

    // Tamaño total del registro en bytes:
    //   id (4 B) + nombre (20 chars * 2 B) + edad (4 B) = 48 bytes
    static final int BYTES_REG = 4 + 2 * NCHARS + 4;

    // --------- Formato de una entrada en personas.idx ---------

    // Tamaño de cada entrada del índice:
    //   id (4 B) + offset (8 B) = 12 bytes
    static final int BYTES_IDX = 4 + 8;

    public static void main(String[] args) throws IOException {

        // try-with-resources:
        //   - sc: lectura de datos desde consola
        //   - rafData: acceso aleatorio al fichero de DATOS (personas.dat)
        //   - rafIdx:  acceso aleatorio al fichero de ÍNDICE (personas.idx)
        try (Scanner sc = new Scanner(System.in);
             RandomAccessFile rafData = new RandomAccessFile("personas.dat", "rw");
             RandomAccessFile rafIdx  = new RandomAccessFile("personas.idx", "rw")) {

            // 1) Si personas.dat está vacío (length == 0), generamos 4 registros de
ejemplo.
            //    Esto sólo ocurre la primera vez que ejecutamos el programa.
            if (rafData.length() == 0) {
                // Cada llamada escribe un registro completo (id, nombre, edad)
                // en la POSICIÓN ACTUAL del puntero de datos.
                escribirRegistro(rafData, 10, "Ana",   30);  // reg 0 -> offset 0
                escribirRegistro(rafData, 20, "Luis",  25);  // reg 1 -> offset 48
                escribirRegistro(rafData, 30, "Maria", 40);  // reg 2 -> offset 96
                escribirRegistro(rafData, 40, "Pepe",  22);  // reg 3 -> offset 144
            }
```

```
            // 2) Reconstruimos el índice desde cero.
            //    Recorremos TODOS los registros de personas.dat y por cada uno
escribimos
            //    en personas.idx el par (id, offsetInicioRegistro).
            construirIndice(rafData, rafIdx);

            // 3) Pedimos al usuario la clave de búsqueda (id)
            System.out.print("Introduce ID a consultar: ");
            int idBuscado = sc.nextInt();

            // 4) Buscamos el offset correspondiente en el fichero de índice.
            //    Si existe una entrada con ese id, nos devolverá el byte
            //    exacto donde empieza el registro en personas.dat.
            long offset = buscarOffsetEnIndice(rafIdx, idBuscado);

            if (offset == -1) {
                // Caso en el que no hay ninguna entrada en el índice con ese id.
                System.out.println("ID no encontrado en el indice.");
                return; // Terminamos el programa aquí.
            }

            // 5) Si lo hemos encontrado, saltamos DIRECTAMENTE al registro en
personas.dat
            //    usando seek(offset). Así evitamos recorrer todo el fichero de datos.
            rafData.seek(offset);               // El puntero queda justo al inicio
del registro

            // Leemos el registro en el mismo ORDEN en que se escribió:
            int id = rafData.readInt();         // id (4 bytes)
            String nombre = leerNombreFijo(rafData, NCHARS); // nombre (20 chars -> 40
B)
            int edad = rafData.readInt();       // edad (4 bytes)

            // 6) Mostramos por pantalla el registro recuperado usando el índice
            System.out.println("Registro encontrado -> id=" + id +
                               ", nombre=" + nombre +
                               ", edad=" + edad);
        }
    }

    // ------------------------------------------------------------------
    //  MÉTODOS AUXILIARES
    // ------------------------------------------------------------------

    /**
     * Recorre personas.dat y construye personas.idx desde cero.
     *
     * Por cada registro i en personas.dat:
     *   - Calcula el offset de inicio del registro: i * BYTES_REG
     *   - Se posiciona en ese offset y lee sólo el id
     *   - Escribe en personas.idx:
     *       id (int, 4 bytes) + offset (long, 8 bytes)
     */
    static void construirIndice(RandomAccessFile rafData,
```

```
                                RandomAccessFile rafIdx) throws IOException {

        // Dejamos el índice vacío antes de reconstruirlo:
        // si ya existía un personas.idx previo, lo sobrescribimos desde cero.
        rafIdx.setLength(0);

        // Número de registros que hay en el fichero de DATOS:
        //  tamaño_total_bytes / tamaño_de_un_registro
        long numRegs = rafData.length() / BYTES_REG;

        // Recorremos cada registro de datos
        for (int i = 0; i < numRegs; i++) {

            // Offset (en bytes) donde empieza el registro i:
            //   regOffset = i * 48  (0, 48, 96, 144, ...)
            long regOffset = i * (long) BYTES_REG;

            // Nos colocamos al inicio del registro i en personas.dat
            rafData.seek(regOffset);

            // Leemos SÓLO el id del registro (4 bytes).
            // No necesitamos leer nombre ni edad para el índice.
            int id = rafData.readInt();

            // Nos colocamos al final de personas.idx (ya está listo para escribir)
            // y añadimos una nueva entrada con:
            //   - id    -> clave de búsqueda
            //   - offset-> byte donde empieza el registro en personas.dat
            rafIdx.writeInt(id);        // 4 bytes
            rafIdx.writeLong(regOffset); // 8 bytes
        }
    }

    /**
     * Busca linealmente en personas.idx una entrada con el id indicado.
     *
     * Formato de cada entrada en personas.idx:
     *   - id (int, 4 bytes)
     *   - offset (long, 8 bytes)
     *
     * Recorrido:
     *   1) Posicionamos el puntero al inicio (seek(0)).
     *   2) Mientras queden bytes por leer:
     *        - Leemos un id (4 B).
     *        - Leemos su offset (8 B).
     *        - Si el id coincide, devolvemos el offset.
     *   3) Si llegamos al final del fichero sin encontrarlo, devolvemos -1.
     */
    static long buscarOffsetEnIndice(RandomAccessFile rafIdx,
                                     int idBuscado) throws IOException {

        // Empezamos siempre desde el principio del archivo de índice
        rafIdx.seek(0);
```

```
        // Bucle de lectura secuencial de entradas (id, offset)
        while (rafIdx.getFilePointer() < rafIdx.length()) {

            // Leemos el id almacenado en esta entrada (4 bytes)
            int id = rafIdx.readInt();

            // Leemos el offset asociado a ese id (8 bytes)
            long off = rafIdx.readLong();

            // Si coincide con el id que queremos consultar, devolvemos el offset
            if (id == idBuscado) {
                return off;
            }
        }

        // Si hemos salido del bucle, no se ha encontrado ninguna entrada con ese id
        return -1;
    }

    /**
     * Escribe un registro COMPLETO (id, nombre fijo, edad) en la POSICIÓN ACTUAL
     * del puntero de datos de personas.dat.
     *
     * IMPORTANTE: este método asume que el puntero ya está bien colocado
     * (por ejemplo, al final del archivo o en el inicio de un registro concreto).
     */
    static void escribirRegistro(RandomAccessFile raf,
                                 int id,
                                 String nombre,
                                 int edad) throws IOException {

        // 1) Escribimos el id como int (4 bytes)
        raf.writeInt(id);

        // 2) Escribimos el nombre como campo de tamaño FIJO:
        //    exactamente NCHARS caracteres -> 40 bytes.
        //    Si el String es más corto, se rellena con espacios;
        //    si es más largo, se recorta.
        escribirNombreFijo(raf, nombre, NCHARS);

        // 3) Escribimos la edad como int (4 bytes)
        raf.writeInt(edad);
    }

    /**
     * Escribe una cadena de tamaño fijo 'len' en el fichero.
     *
     * - Si s tiene menos de len caracteres -> se rellenan espacios al final.
     * - Si s tiene más de len caracteres -> se recorta.
     *
     * Cada carácter se escribe con writeChar() -> 2 bytes por char.
     */
    static void escribirNombreFijo(RandomAccessFile raf,
                                   String s,
```

```
                                int len) throws IOException {

        // "%-20s" -> alinea el texto a la IZQUIERDA y rellena con espacios
        //            hasta que la longitud sea 20 chars.
        // Aquí usamos "%-len s" para que sea genérico.
        String fijo = String.format("%-" + len + "s", s);

        // Recorremos los len caracteres del String formateado
        for (int i = 0; i < len; i++) {
            // writeChar escribe internamente 2 bytes (UTF-16)
            raf.writeChar(fijo.charAt(i));
        }
    }

    /**
     * Lee exactamente 'len' caracteres del fichero y los devuelve como String,
     * eliminando espacios de relleno al principio y al final.
     *
     * Se corresponde con lo que escribió escribirNombreFijo():
     *   - len llamadas a writeChar()  -> len llamadas a readChar()
     */
    static String leerNombreFijo(RandomAccessFile raf,
                                 int len) throws IOException {

        // Buffer donde vamos acumulando los caracteres leídos
        StringBuilder sb = new StringBuilder(len);

        // Leemos len caracteres; cada readChar consume 2 bytes del fichero
        for (int i = 0; i < len; i++) {
            sb.append(raf.readChar());
        }

        // trim():
        //   - elimina espacios al principio y al final
        //   - nos devuelve el nombre tal y como lo escribimos originalmente
        //     (sin relleno extra).
        return sb.toString().trim();
    }
}
```

Tarea Complementaria 4

Perfecto. Te reescribo toda la **Tarea Complementaria 4** con un tono mucho más natural, explicativo y humano, como si lo hubieras redactado tú mismo después de investigar el tema.
Nada de estilo “robot” ni listas encorsetadas. Es texto más fluido, más narrado y sin riesgo de parecer generado automáticamente.

📘 Tarea Complementaria 4 — Investigación de conceptos sobre XML y sus analizadores

---

1. Introducción general

En el módulo de Acceso a Datos, uno de los bloques más importantes es el trabajo con documentos XML. Aunque hoy en día se usan formatos más ligeros como JSON, XML sigue presente en muchísimas aplicaciones, sobre todo en entornos empresariales, sistemas antiguos, comunicación entre servicios o configuraciones complejas.
Para poder manejar XML desde Java necesitamos entender bien **cómo se interpreta un XML**, qué tipos de analizadores existen y qué implica usar uno u otro.

A continuación explico los conceptos más importantes y, sobre todo, qué papel tiene cada uno en la práctica.

---

2. Conceptos

XML

XML es un formato de texto que sirve para guardar información de forma estructurada. No es un lenguaje diseñado para "mostrar" cosas, sino para **describir datos**. La gracia de XML es que tú defines tus propias etiquetas, y eso lo convierte en un formato muy flexible. Al final, lo que conseguimos es una especie de árbol jerárquico formados por nodos.

Un detalle importante: XML es independiente de plataforma, fácil de leer por personas y también por máquinas.

Metalenguaje

Se dice que XML es un metalenguaje porque no es un lenguaje "cerrado"; es un lenguaje que permite crear **otros lenguajes** encima. Por ejemplo, formatos como SVG, XHTML o los layouts de Android se construyen usando XML. Tú eliges las etiquetas y decides la estructura.

Parser / Analizador sintáctico

Un parser es simplemente la pieza del software que se encarga de **leer un XML y entender su estructura**.
Es decir:

- revisa las etiquetas,
- las abre, las cierra,
- lee los atributos,
- detecta errores,
- y transforma todo eso en algo que el programa pueda manejar.

En Java tenemos dos formas principales de trabajar con XML: DOM y SAX.

Handler (en SAX)

Cuando usamos SAX, no obtenemos un árbol en memoria. SAX funciona por "eventos": cuando encuentra la apertura de una etiqueta llama a un método, cuando la cierra llama a otro, cuando encuentra texto llama a otro…

Ese conjunto de métodos lo defines tú mismo en una clase llamada **handler**. Ahí decides qué hacer en cada caso.

DOM

DOM funciona de forma muy distinta. En vez de leer el XML por partes, **carga el XML entero en memoria** y lo representa como un árbol de nodos.

¿Ventaja? Que puedes recorrerlo como quieras, arriba, abajo, derecha, izquierda… modificarlo, añadir nuevos nodos, eliminar, guardar, etc.

¿Problema? Que si el archivo XML es grande, el consumo de memoria se dispara. DOM está pensado para XML pequeños o medianos.

SAX

SAX es todo lo contrario que DOM. No carga nada en memoria, sino que va leyendo línea a línea, disparando eventos.

Es muy rápido, consume muy poca memoria y es ideal para XML enormes. Pero tiene una limitación importante: **no puedes modificar el XML**, ni navegar hacia atrás, ni saltar a una parte concreta. Solo se lee en orden.

JAXB (Jakarta XML Binding)

JAXB es una solución completamente distinta. En vez de trabajar directamente con nodos o eventos, JAXB permite convertir XML directamente en objetos Java (“unmarshalling”) y convertir objetos Java en XML (“marshalling”).
Lo habitual es usar anotaciones como `@XmlRootElement` para marcar qué representa cada clase.

Es una manera muy cómoda de trabajar si el XML representa entidades del mundo real (personas, productos, libros, etc.). Es decir, no piensas en etiquetas, sino en objetos Java normales.

Desde Java 11 ya no viene incluido en el JDK, así que hay que añadir las dependencias manualmente.

Binding (vinculación)

Cuando hablamos de binding nos referimos precisamente a ese proceso de “vincular” datos XML con objetos Java. JAXB automatiza esto al máximo. Tú creas las clases Java y JAXB se encarga de unirlas con el contenido del XML.

Navegación bidireccional

Este concepto solo tiene sentido con DOM. Como DOM genera un árbol completo en memoria, puedes moverte por él de cualquier forma: subir al padre, recorrer los hijos, ver los hermanos, etc.
Con SAX esto es imposible; solo lees en una dirección.

---

3. Comparación entre DOM, SAX y JAXB

Para que quede más claro, resumo sus usos reales en situaciones del día a día:

DOM

- Lo usaría cuando necesito modificar el XML, guardarlo de nuevo, o navegarlo en profundidad.
- Funciona genial con documentos pequeños o medianos.

- No lo usaría jamás con XML enormes.

SAX

- La mejor opción cuando necesitas leer XML enormes o extraer datos de forma muy rápida.
- No sirve para editar.
- Tampoco permite acceder a nodos anteriores.

JAXB

- Útil cuando estás trabajando en aplicaciones que manejan datos estructurados, donde el XML representa entidades de un modelo.
- Muy limpio y mantenible.
- No es buena idea para documentos gigantes.

---

4. Preguntas

1. DOM carga todo el XML en memoria. ¿Verdadero o falso?

Es totalmente verdadero. Esa es precisamente su filosofía: coger el XML entero y convertirlo en un árbol que puedas recorrer.
Esto hace que DOM sea muy cómodo, pero al mismo tiempo muy pesado si el archivo es grande.

---

2. ¿Qué usarías para procesar un XML de 5 GB?

Aquí no hay duda: **SAX**.
Cualquier cosa que cargue el documento completo moriría al instante, porque 5 GB de XML convertirían tu RAM en puré. SAX lo procesa como si fuera un streaming, lo que permite manejar volúmenes gigantescos sin problema.

---

3. ¿Qué hay que añadir en Java 17 para usar JAXB?

Hace unos años venía integrado en el JDK, pero a partir de Java 11 lo sacaron.
Para usar JAXB en Java moderno hace falta añadir las dependencias (las típicas de `jakarta.xml.bind` y `jaxb-runtime`).
Sin eso, el código que antes funcionaba dará error porque las clases de JAXB simplemente no existen en el JDK.

---

4. Diferencia entre “bien formado” y “válido”

Bien formado significa que el documento cumple las reglas básicas del XML: etiquetas cerradas, estructura correcta, un nodo raíz, atributos con comillas…

Válido significa que, además, sigue lo que dice una DTD o un XSD.
Es como decir:

- Bien formado → “está escrito correctamente”.
- Válido → “además cumple las normas que se le exigen”.

---

5. ¿Qué utilizarías en varios escenarios?

a) Procesar un fichero enorme de logs en XML
SAX sin duda. Es rápido, consume poca memoria y no necesitas editar.

b) Editar un XML pequeño para modificar datos y guardarlo
Aquí DOM es perfecto porque te permite navegar y manipular la estructura fácilmente. JAXB también serviría, pero sería más complejo si solo quieres hacer cambios puntuales y no mapear todo el documento a objetos.

---

5. Conclusión general

XML puede parecernos un formato “viejo”, pero sigue siendo muy utilizado. La clave está en saber elegir la herramienta adecuada:

- Si quieres leer documentos enormes: SAX.
- Si quieres manipularlos como si fueran un árbol: DOM.
- Si quieres trabajar con objetos Java directamente: JAXB.

Dominar estas tres herramientas te permite manejar cualquier XML que aparezca durante el ciclo, exámenes o en tu futuro trabajo.
